Aug 88

The C Preprocessor

First Edition

January 1987

Richard M. Stallman

Copyright © 1987 Richard M. Stallman.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1. The C Preprocessor 1
1.1 Transformations Made Globally 1
1.2 Preprocessor Commands 3
1.3 Header Files 3
1.3.1 Uses of Header Files 3
1.3.2 The '`#include`' Command 4
1.3.3 How '`#include`' Works 5
1.4 Macros 6
1.4.1 Simple Macros 6
1.4.2 Macros with Arguments 8
1.4.3 Predefined Macros 10
1.4.3.1 Standard Predefined Macros 10
1.4.3.2 Nonstandard Predefined Macros 12
1.4.4 Stringification 13
1.4.5 Concatenation 14
1.4.6 Undefining Macros 16
1.4.7 Redefining Macros 17
1.4.8 Pitfalls and Subtleties of Macros 17
1.4.8.1 Improperly Nested Constructs 17
1.4.8.2 Unintended Grouping of Arithmetic 18
1.4.8.3 Swallowing the Semicolon 19
1.4.8.4 Duplication of Side Effects 21
1.4.8.5 Self-Referential Macros 22
1.4.8.6 Separate Expansion of Macro Arguments 23
1.4.8.7 Cascaded Use of Macros 24
1.5 Conditionals 25
1.5.1 Syntax of Conditionals 26
1.5.1.1 The '`#if`' Command 26
1.5.1.2 The '`#else`' Command 27
1.5.1.3 The '`#elif`' Command 27
1.5.2 Keeping Deleted Code for Future Reference 28
1.5.3 Conditionals and Macros 29
1.5.4 The '`#error`' Command 30
1.6 Combining Source Files 31
1.7 Miscellaneous Preprocessor Commands 32
1.8 C Preprocessor Output 32
1.9 Invoking the C Preprocessor 32

Concept Index 35

Index of Commands, Macros and Options 36

1. The C Preprocessor

The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessor commands, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, the C Compatible Compiler Preprocessor. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options '`-T`', '`-undef`' and '`-pedantic`', but in practice the consequences of having strict ANSI Standard C make it undesirable to do this. See section 1.9 [Invocation], page 32.

1.1 Transformations Made Globally

Most C preprocessor features are inactive unless you give specific commands to request their use. (Preprocessor commands are lines starting with '`#`'; see section 1.2 [Commands], page 3). But there are three transformations that the preprocessor always makes on all the input it receives, even in the absence of commands.

- All C comments are replaced with single spaces.

- Backslash-Newline sequences are deleted, no matter where. This feature allows you to break long lines for cosmetic purposes without changing their meaning.
- Predefined macro names are replaced with their expansions (see section 1.4.3 [Predefined], page 10).

The first two transformations are done *before* nearly all other parsing and before preprocessor commands are recognized. Thus, for example, you can split a line cosmetically with Backslash-Newline anywhere (except when trigraphs are in use; see below).

```
/*
*/ # /*
*/ defi\
ne FO\
O 10\
20
```

is equivalent into '`#define FOO 1020`'. You can split even an escape sequence with Backslash-Newline. For example, you can split `"foo\bar"` between the '`\`' and the '`b`' to get

```
"foo\\
bar"
```

This behavior is unclean: in all other contexts, a Backslash can be inserted in a string constant as an ordinary character by writing a double Backslash, and this creates an exception. But the ANSI C standard requires it. (Strict ANSI C does not allow Newlines in string constants, so they do not consider this a problem.)

But there are a few exceptions to all three transformations.

- C comments and predefined macro names are not recognized inside a '`#include`' command in which the file name is delimited with '`<`' and '`>`'.
- C comments and predefined macro names are never recognized within a character or string constant. (Strictly speaking, this is the rule, not an exception, but it is worth noting here anyway.)
- Backslash-Newline may not safely be used within an ANSI "trigraph". Trigraphs are converted before Backslash-Newline is deleted. If you write what looks like a trigraph with a Backslash-Newline inside, the Backslash-Newline is deleted as usual, but it is then too late to recognize the trigraph.

  This exception is relevant only if you use the '`-T`' option to enable trigraph processing. See section 1.9 [Invocation], page 32.

1.2 Preprocessor Commands

Most preprocessor features are active only if you use preprocessor commands to request their use.

Preprocessor commands are lines in your program that start with '#'. The '#' is followed by an identifier that is the *command name*. For example, '`#define`' is the command that defines a macro. Whitespace is also allowed before and after the '#'.

The set of valid command names is fixed. Programs cannot define new preprocessor commands.

Some command names require arguments; these make up the rest of the command line and must be separated from the command name by whitespace. For example, '`#define`' must be followed by a macro name and the intended expansion of the macro.

A preprocessor command cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the command into multiple lines, but the comments are changed to Spaces before the command is interpreted. The only way a significant Newline can occur in a preprocessor command is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The '#' and the command name cannot come from a macro expansion. For example, if '`foo`' is defined as a macro expanding to '`define`', that does not make '`#foo`' a valid preprocessor command.

1.3 Header Files

A header file is a file containing C declarations and macro definitions (see section 1.4 [Macros], page 6) to be shared between several source files. You request the use of a header file in your program with the C preprocessor command '`#include`'.

1.3.1 Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with '`.h`'.

1.3.2 The '`#include`' Command

Both user and system header files are included using the preprocessor command '`#include`'. It has three variants:

`#include <`*file*`>`

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option '`-I`' (see section 1.9 [Invocation], page 32). The option '`-nostdinc`' inhibits searching the standard system directories; in this case only the directories you specify are searched.

The parsing of this form of '`#include`' is slightly special because comments are not recognized within the '`<...>`'. Thus, in '`#include <x/*y>`' the '`/*`' does not start a comment and the command specifies inclusion of a system header file named '`x/*y`'. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate.

The argument *file* may not contain a '`>`' character. It may, however, contain a '`<`' character.

`#include "`*file*`"`

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header

files. The current directory is tried first because it is presumed to be the location of the files of the program being compiled. (If the '`-I-`' option is used, the special treatment of the current directory is inhibited.)

The argument *file* may not contain '"' characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, '`#include "x\n\\y"`' specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#include` *anything else*

This variant is called a *computed #include*. Any '`#include`' command whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded (see section 1.4 [Macros], page 6). When this is done, the result must fit one of the above two variants.

This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

1.3.3 How '`#include`' Works

The '`#include`' command works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the '`#include`' command. For example, given two files as follows:

```
/* File program.c */
int x;
#include "header.h"

main ()
{
  printf (test ());
}
```

```
/* File header.h */
char *test ();
```

the output generated by the C preprocessor for '`program.c`' as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; they are merely the typical use. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

The line following the '`#include`' command is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

1.4 Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

1.4.1 Simple Macros

A *simple macro* is a kind of abbreviation. It is a name which stands for a fragment of code.

Before you can use a macro, you must *define* it explicitly with the '`#define`' command. '`#define`' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named '`BUFFER_SIZE`' as an abbreviation for the text '`1020`'. Therefore, if somewhere after this '`#define`' command there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and *expand* the macro 'BUFFER_SIZE', resulting in

```
foo = (char *) xmalloc (1020);
```

the definition must be a single line; however, it may not end in the middle of a multi-line string constant or character constant.

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessor commands. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. By the same token, it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above. there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (Of course, you might get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body

can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name 'TABLESIZE' when used in the program would go through two stages of expansion, resulting ultimately in '1020'.

This is not at all the same as defining 'TABLESIZE' to be '1020'. The '#define' for 'TABLESIZE' uses exactly the body you specify—in this case, 'BUFSIZE'—and does not check to see whether it too is the name of a macro. It's only when you *use* 'TABLESIZE' that the result of its expansion is checked for more macro names. See section 1.4.8.7 [Cascaded Macros], page 24.

1.4.2 Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept *arguments*. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition.

To define a macro that uses arguments, you write a '#define' command with a list of *argument names* in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a "minimum" macro in GNU C. See section 1.4.8.4 [Side Effects], page 21, for more information.)

To use a macro that expects arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses. separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro 'min' include 'min (1, 2)' and 'min (x + 28, *p)'.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro '`min`' defined above, '`min (1, 2)`' expands into

```
((1) < (2) ? (1) : (2))
```

where '`1`' has been substituted for '`X`' and '`2`' for '`Y`'.

Likewise, '`min (x + 28, *p)`' expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance; thus, if you want to supply '`array[x = y, x + 1]`' as an argument, you must write it as '`array[(x = y, x + 1)]`', which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, '`min (min (a, b), c)`' expands into

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name

for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named 'min' in the same source file that defines the macro. If you write '&min' with no argument list, you refer to the function. If you write 'min (x, bb)', with an argument list, the macro is expanded. If you write '(min) (a, bb)', where the name 'min' is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function 'min'.

It is not allowed to define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the name is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines 'FOO' to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines 'BAR' to take no argument and always expand into '(x) - 1 / (x)').

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

1.4.3 Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

1.4.3.1 Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double

underscores. Those preceding `__GNUC__` in this table are standardized by ANSI C; the rest are GNU C extensions.

`__FILE__` This macro expands to the name of the current input file, in the form of a C string constant.

`__LINE__` This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

This and '`__FILE__`' are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: negative string length "
                 "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A '`#include`' command changes the expansions of '`__FILE__`' and '`__LINE__`' to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the '`#include`' command, the expansions of '`__FILE__`' and '`__LINE__`' revert to the values they had before the '`#include`' (but '`__LINE__`' is then incremented by one as processing moves to the line after the '`#include`').

The expansions of both '`__FILE__`' and '`__LINE__`' are altered if a '`#line`' command is used. See section 1.6 [Combining Sources], page 31.

`__DATE__` This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like '`"Jan 29 1987"`' or '`"Apr 1 1905"`'.

`__TIME__` This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like '`"23:59:01"`'.

`__STDC__` This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

`__GNUC__` This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, '`__GNUC__`' is undefined.

`__STRICT_ANSI__`
This macro is defined if and only if the '`-ansi`' switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

`__VERSION__`

This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as `"1.18"`. The only reasonable use of this macro is to incorporate it into a string constant.

`__OPTIMIZE__`

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

`__CHAR_UNSIGNED__`

This macro is defined if and only if the data type `char` is unsigned on the target machine. It exists to cause the standard header file '`limit.h`' to work correctly. It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in '`limit.h`'.

1.4.3.2 Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones.

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

`unix` '`unix`' is normally predefined on all Unix systems.

`BSD` '`BSD`' is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax` '`vax`' is predefined on Vax computers.

`mc68000` '`mc68000`' is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k` '`m68k`' is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use '`mc68000`' and some use '`m68k`'. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020` '`M68020`' has been observed to be predefined on some systems that use 68020 CPUs—in addition to '`mc68000`' and '`m68k`' that are less specific.

`ns32000` 'ns32000' is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun` 'sun' is predefined on all models of Sun computers.

`pyr` 'pyr' is predefined on all models of Pyramid computers.

`sequent` 'sequent' is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. However, the GNU C preprocessor would be useless if it did not predefine the same names that are normally predefined on the system and machine you are using. Even system header files check the predefined names and will generate incorrect declarations if they do not find the names that are expected.

The set of nonstandard predefined names in the GNU C preprocessor is controlled by the macro 'CPP_PREDEFINES', which should be a string containing '-D' options, separated by spaces. For example, on the Sun, the definition

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

is used.

The '-ansi' option which requests complete support for ANSI C inhibits the definition of these predefined symbols.

1.4.4 Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying 'foo (z)' results in '"foo (z)"'.

In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character '#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no '#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) fprintf (stderr, "Warning: " #EXP "\n"); } while (0)
```

Here the actual argument for 'EXP' is substituted once as given, into the 'if' statement, and once as stringified, into the argument to 'fprintf'. The 'do' and 'while (0)' are a kludge to make it possible to write 'WARN_IF (arg);', which the resemblance of 'WARN_IF' to a function would make C programmers want to do; see section 1.4.8.3 [Swallow Semicolon], page 19).

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies 'EXP''s actual argument into a separate string constant, resulting in text like

```
do { if (x == 0) fprintf (stderr, "Warning: " "x == 0" "\n"); } while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) fprintf (stderr, "Warning: x == 0\n"); } while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying 'p = "foo\n";' results in '"p = \"foo\\n\";"'. However, backslashes that are not inside of string or character constants are not duplicated: '\n' by itself stringifies to '"\n"'.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

1.4.5 Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer

name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator '##' in the macro body. When the macro is called, after actual arguments are substituted, all '##' operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the '##'.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with '`_command`'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as '`1.5`' and '`e3`') into a number. Also, multi-character operators such as '`+=`' can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot

be concatenated. For example, concatenation with '`x`' on one side and '`+`' on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the '`x`' and '`+`' side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating '`/`' and '`*`': the '`/*`' sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a '`##`' in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

1.4.6 Undefining Macros

To *undefine* a macro means to cancel its definition. This is done with the '`#undef`' command. '`#undef`' is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, '`FOO`' had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of '`#undef`' command will cancel definitions with arguments or definitions that

don't expect arguments. The '`#undef`' command has no effect when used on a name not currently defined as a macro.

1.4.7 Redefining Macros

Redefining a macro means defining (with '`#define`') a name that is already defined as a macro.

A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once (see section 1.3 [Header Files], page 3), so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with '`#undef`' before the second definition.

In order for a reefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

1.4.8 Pitfalls and Subtleties of Macros

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counterintuitive consequences that you must watch out for.

1.4.8.1 Improperly Nested Constructs

Recall that when a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls.

It is possible to piece together a macro call coming partially from the macro body and partially from the actual arguments. For example,

```
#define double(x) (2*(x))
#define call_with_1(x) x(1)
```

would expand '`call_with_1 (double)`' into '`(2*(1))`'.

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
```

This bizarre example expands to '`fprintf (stderr, "%s %d", p, 35)`'!

1.4.8.2 Unintended Grouping of Arithmetic

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro argument name had parentheses around it. In addition, another pair of parentheses usually surround the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many '`int`"s are needed to hold a certain number of '`char`"s.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
```

This expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

But what we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

However, unintended grouping can result in another way. Consider '`sizeof ceil_div(1, 2)`'. That has the appearance of a C expression that would compute the size of the type of '`ceil_div (1, 2)`', but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the '`sizeof`' when it was intended to be inside.

Parentheses around the entire macro definition can prevent such problems. Here, then, is the recommended way to define '`ceil_div`':

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

1.4.8.3 Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the argument '`p`' says where to find it) across whitespace characters:

```
#define SKIP_SPACES (p, limit)  \
{ register char *lim = (limit); \
  while (p != lim) {            \
```

```
if (*p++ != ' ') {          \
  p--; break; }}}
```

Here Backslash-Newline is used to split the macro definition, which must be a single line, so that it resembles the way such C code would be layed out if not part of a macro definition.

A call to this macro might be '`SKIP_SPACES (p, lim)`'. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in '`SKIP_SPACES (p, lim);`'

But this can cause trouble before '`else`' statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
else ...
```

The presence of two statements—the compound statement and a null statement—in between the '`if`' condition and the '`else`' makes invalid C code.

The definition of the macro '`SKIP_SPACES`' can be altered to solve this problem, using a '`do ... while`' statement. Here is how:

```
#define SKIP_SPACES (p, limit)     \
do { register char *lim = (limit); \
     while (p != lim) {            \
       if (*p++ != ' ') {          \
         p--; break; }}}           \
while (0)
```

Now '`SKIP_SPACES (p, lim);`' expands into

```
do {...} while (0);
```

which is one statement.

1.4.8.4 Duplication of Side Effects

Many C programs define a macro 'min', for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where 'x + y' has been substituted for 'X' and 'foo (z)' for 'Y'.

The function 'foo' is used only once in the statement as it appears in the program, but the expression 'foo (z)' has been substituted twice into the macro expansion. As a result, 'foo' might be called two times when the statement is executed. If it has side effects or if it takes a long time to compute, the results might not be what you intended. We say that 'min' is an *unsafe* macro.

The best solution to this problem is to define 'min' in a way that computes the value of 'foo (z)' only once. The C language offers no way standard way to do this, but it can be done with GNU C extensions as follows:

```
#define min(X, Y)                       \
({ typeof (X) __x = (X), __y = (Y);     \
   (__x < __y) ? __x : __y; })
```

If you do not wish to use GNU C extensions, the only solution is to be careful when *using* the macro 'min'. For example, you can calculate the value of 'foo (z)', save it in a variable, and use that variable in 'min':

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
...
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where I assume that '`foo`' returns type '`int`').

1.4.8.5 Self-Referential Macros

A *self-referential* macro is one whose name appears in its definition. A special feature of ANSI Standard C is that the self-reference is not considered a macro call. It is passed into the preprocessor output unchanged.

Let's consider an example:

```
#define foo (4 + foo)
```

where '`foo`' is also a variable in your program.

Following the ordinary rules, each reference to '`foo`' will expand into '`(4 + foo)`'; then this will be rescanned and will expand into '`(4 + (4 + foo))`'; and so on until it causes a fatal error (memory full) in the preprocessor.

However, the special rule about self-reference cuts this process short after one step, at '`(4 + foo)`'. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of '`foo`' wherever '`foo`' is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that '`foo`' is a variable will not expect that it is a macro as well. The reader will come across a the identifier '`foo`' in the program and think its value should be that of the variable '`foo`', whereas in fact the value is four greater.

The special rule for self-reference applies also to *indirect* self-reference. This is the case where a macro `x` expands to use a macro '`y`', and '`y`''s expansion refers to the macro '`x`'. The resulting reference to '`x`' comes indirectly from the expansion of '`x`', so it is a self-reference and is not further expanded. Thus, after

```
#define x (4 + y)
#define y (2 * x)
```

'`x`' would expand into '`(4 + (2 * x))`'. Clear?

But suppose 'y' is used elsewhere, not from the definition of 'x'. Then the use of 'x' in the expansion of 'y' is not a self-reference because 'x' is not "in progress". So it does expand. However, the expansion of 'x' contains a reference to 'y', and that is an indirect self-reference now because 'y' is "in progress". The result is that 'y' expands to '`(2 * (4 + y))`'.

It is not clear that this behavior would ever be useful, but it is specified by the ANSI C standard, so you need to understand it.

1.4.8.6 Separate Expansion of Macro Arguments

We have explained that the expansion of a macro, including the substituted actual arguments, is scanned over again for macro calls to be expanded.

What really happens is more subtle: first each actual argument text is scanned separately for macro calls. Then the results of this are substituted into the macro body to produce the macro expansion, and the macro expansion is scanned again for macros to expand.

The result is that the actual arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the actual argument contained any macro calls, they are expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the actual argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an actual argument of another macro (see section 1.4.8.5 [Self-Reference], page 22): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. But this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

The prescan is not done when an argument is stringified or concatenated. (More precisely, stringification and concatenation use the argument as written, in un-prescanned form. The same actual argument would be used in prescanned form if it is substituted elsewhere without stringification or concatenation.) Thus,

```
#define str(s) #s
#define foo 4
str (foo)
```

expands to '"foo"'. Once more, prescan has been prevented from having any noticeable effect.

You might now ask, "Why mention the prescan, if it makes no difference? Why not skip it and make the preprocessor faster?" The answer is that the prescan does make a difference in two special cases:

- Nested calls to a macro.
- Macros that call other macros that stringify or concatenate.

We say that *nested* calls to a macro occur when a macro's actual argument contains a call to that very macro. For example, if 'f' is a macro that expects one argument, 'f (f (1))' is a nested pair of calls to 'f'. The desired expansion is made by expanding 'f (1)' and substituting that into the definition of 'f'. The prescan causes the expected result to happen. Without the prescan, 'f (1)' itself would be substituted as an actual argument, and the inner use of 'f' would appear during the main scan as an indirect self-reference and would not be expanded. Here, the prescan cancels an undesirable side effect (in the medical, not computational, sense of the term) of the special rule for self-referential macros.

There is also one case where prescan is useful. It is possible to use prescan to expand an argument and then stringify it—if you use two levels of macros. Let's add a new macro 'xstr' to the example shown above:

```
#define xstr(s) str(s)
#define str(s) #s
#define foo 4
xstr (foo)
```

This expands into '"4"', not '"foo"'. The reason for the difference is that the argument of 'xstr' is expanded at prescan (because 'xstr' does not specify stringification or concatenation of the argument). The result of prescan then forms the actual argument for 'str'. 'str' uses its argument without prescan because it performs strigification; but it cannot prevent or undo the prescanning already done by 'xstr'.

1.4.8.7 Cascaded Use of Macros

A *cascade* of macros is when one macro's body contains a reference to another macro. This is very common practice. For example,

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

This is not at all the same as defining 'TABLESIZE' to be '1020'. The '#define' for 'TABLESIZE' uses exactly the body you specify—in this case, 'BUFSIZE'—and does not check to see whether it too is the name of a macro.

It's only when you *use* 'TABLESIZE' that the result of its expansion is checked for more macro names.

This makes a difference if you change the definition of 'BUFSIZE' at some point in the source file. 'TABLESIZE', defined as shown, will always expand using the definition of 'BUFSIZE' that is currently in effect:

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now 'TABLESIZE' expands (in two stages) to '37'.

1.5 Conditionals

In a macro processor, a *conditional* is a command that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an 'if' statement in C, but it is important to understand the difference between them. The condition in an 'if' statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessor conditional command is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Generally there are three kinds of reason to use a conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another

operating system; for example, it might refer to library routines that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code: merely having it in the program makes it impossible to link the program and run it. With a preprocessor conditional, the offending code can be effectively excised from the program when it is not valid.

- You may want to be able to compile the same source file into two different programs. Sometimes the difference between the programs is that one makes frequent time-consuming consistency checks on its intermediate data while the other does not.
- A conditional whose condition is always false is a good way to exclude code from the program but keep it as a sort of comment for future reference.

Most simple programs that are intended to run on only one machine will not need to use preprocessor conditionals.

1.5.1 Syntax of Conditionals

A conditional in the C preprocessor begins with a *conditional command*: '`#if`', '`#ifdef`' or '`#ifndef`'.See section 1.5 [Conditionals], page 25, for info on '`#ifdef`' and '`#ifndef`'; only '`#if`' is explained here.

1.5.1.1 The '`#if`' Command

The '`#if`' command in its simplest form consists of

```
#if expression
controlled text
#endif /* expression */
```

The comment following the '`#endif`' is not required, but it is a good practice because it helps people match the '`#endif`' to the corresponding '`#if`'. Such comments should always be used, except in short conditionals that are not nested. In fact, you can put anything at all after the '`#endif`' and it will be ignored by the GNU C preprocessor, but only comments are acceptable in ANSI Standard C.

expression is a C expression of integer type, subject to stringent restrictions. It may contain

- Integer constants, which are all regarded as `long` or `unsigned long`.

- Character constants, which are interpreted according to the character set and conventions of the machine and operating system on which the preprocessor is running. The GNU C preprocessor uses the C data type '`char`' for these character constants; therefore, whether some character codes are negative is determined by the C compiler used to compile the preprocessor. If it treats '`char`' as signed, then character codes large enough to set the sign bit will be considered negative; otherwise, no character code is considered negative.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and '`&&`' and '`||`'.
- Identifiers that are not macros, which are all treated as zero(!).
- Macro calls. All macro calls in the expression are expanded before actual computation of the expression's value begins.

Note that '`sizeof`' operators and `enum`-type values are not allowed. `enum`-type values, like all other identifiers that are not taken as macro calls and expanded, are treated as zero.

The text inside of a conditional can include preprocessor commands. Then the commands inside the conditional are obeyed only if that branch of the conditional succeeds. The text can also contain other conditional groups. However, the '`#if`"s and '`#endif`"s must balance.

1.5.1.2 The '`#else`' Command

The '`#else`' command can be added a conditional to provide alternative text to be used if the condition is false. This looks like

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If *expression* is nonzero, and the *text-if-true* is considered included, then '`#else`' acts like a failing conditional and the *text-if-false* is ignored. Contrariwise, if the '`#if`' conditional fails, the *text-if-false* is considered included.

1.5.1.3 The '`#elif`' Command

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
...
#else /* X != 1 */
#if X == 2
...
#else /* X != 2 */
...
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional command, '`#elif`', allows this to be abbreviated as follows:

```
#if X == 1
...
#elif X == 2
...
#else /* X != 2 and X != 1*/
...
#endif /* X != 2 and X != 1*/
```

'`#elif`' stands for "else if". Like '`#else`', it goes in the middle of a '`#if`'-'`#endif`' pair and subdivides it; it does not require a matching '`#endif`' of its own. Like '`#if`', the '`#elif`' command includes an expression to be tested.

The text following the '`#elif`' is processed only if the original '`#if`'-condition failed and the '`#elif`' condition succeeeds. More than one '`#elif`' can go in the same '`#if`'-'`#endif`' group. Then the text after each '`#elif`' is processed only if the '`#elif`' condition succeeds after the original '`#if`' and any previous '`#elif`"s within it have failed. '`#else`' is equivalent to '`#elif 1`', and '`#else`' is allowed after any number of '`#elif`"s, but '`#elif`' may not follow a '`#else`'.

1.5.2 Keeping Deleted Code for Future Reference

If you replace or delete a part of the program but want to keep the old code around as a comment for future reference, the easy way to do this is to put '`#if 0`' before it and '`#endif`' after it.

This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced '`#if`' and '`#endif`').

1.5.3 Conditionals and Macros

Conditionals are rarely useful except in connection with macros. A '`#if`' command whose expression uses no macros is equivalent to '`#if 1`' or '`#if 0`'; you might as well determine which one, by computing the value of the expression yourself, and then simplify the program. But when the expression uses macros, its value can vary from compilation to compilation.

For example, here is a conditional that tests the expression '`BUFSIZE == 1020`', where '`BUFSIZE`' must be a macro.

```
#if BUFSIZE == 1020
  printf ("Large buffers!\n");
#endif /* BUFSIZE is large */
```

The special operator '`defined`' may be used in '`#if`' expressions to test whether a certain name is defined as a macro. Either '`defined` *name*' or '`defined (`*name*`)`' is an expression whose value is 1 if *name* is defined as macro at the current point in the program, and 0 otherwise. For the '`defined`' operator it makes no difference what the definition of the macro is; all that matters is whether there is a definition. Thus, for example,

```
#if defined (vax) || defined (ns16000)
```

would include the following code if either of the names '`vax`' and '`ns16000`' is defined as a macro.

If a macro is defined and later undefined with '`#undef`', subsequent use of the '`defined`' operator will return 0, because the name is no longer defined. If the macro is defined again with another '`#define`', '`defined`' will recommence returning 1.

Conditionals that test just the definedness of one name are very common, so there are two special short conditional commands for this case. They are

`#ifdef` *name*
: is equivalent to '`#if defined (`*name*`)`'.

`#ifndef` *name*
: is equivalent to '`#if ! defined (`*name*`)`'.

Macro definitions can vary between compilations for several reasons.

- Some macros are predefined on each kind of machine. For example, on a Vax, the name '`vax`'

is a predefined macro. On other machines, it would not be defined.

- Many more macros are defined by system header files. Different systems and machines define different macros, or give them different values. It is useful to test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros are a common way of allowing users to customize a program for different machines or applications. For example, the macro 'BUFSIZE' might be defined in a configuration file for your program that is included as a header file in each source file. You would use 'BUFSIZE' in a preprocessor conditional in order to generate different code depending on the chosen configuration.
- Macros can be defined or undefined with '-D' and '-U' command options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with compiler command options. See section 1.9 [Invocation], page 32.

1.5.4 The '#error' Command

The command '#error' causes the preprocessor to report a fatal error. The rest of the line that follows '#error' is used as the error message.

You would use '#error' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a Vax, you might write

```
#ifdef vax
#error Won't work on Vaxen.  See comments at get_last_object.
#endif
```

See section 1.4.3.2 [Nonstandard Predefined], page 12, for why this works.

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with '#error'. For example,

```
#if HASH_TABLE_SIZE % 2 == 0 || HASH_TABLE_SIZE % 3 == 0 \
    || HASH_TABLE_SIZE % 5 == 0
#error HASH_TABLE_SIZE should not be divisible by a small prime
#endif
```

1.6 Combining Source Files

One of the jobs of the C preprocessor is to inform the C compiler of where each line of C code came from: which source file and which line number.

C code can come from multiple source files if you use '`#include`'; both '`#include`' and the use of conditionals and macros can cause the line number of a line in the preprocessor output to be different from the line's number in the original source file. You will appreciate the value of making both the C compiler (in error messages) and symbolic debuggers such as GDB use the line numbers in your source file.

The C preprocessor builds on this feature by offering a command by which you can control the feature explicitly. This is useful when a file for input to the C preprocessor is the output from another program such as the `bison` parser generator, which operates on another file that is the true source file. Parts of the output from `bison` are generated from scratch, other parts come from a standard parser file. The rest are copied nearly verbatim from the source file, but their line numbers in the `bison` output are not the same as their original line numbers. Naturally you would like compiler error messages and symbolic debuggers to know the original source file and line number of each line in the `bison` output.

`bison` arranges this by writing '`#line`' commands into the output file. '`#line`' is a command that specifies the original line number and source file name for subsequent input in the current preprocessor input file. '`#line`' has three variants:

`#line` *linenum*

Here *linenum* is a decimal integer constant. This specifies that the line number of the following line of input, in its original source file, was *linenum*.

`#line` *linenum filename*

Here *linenum* is a decimal integer constant and *filename* is a string constant. This specifies that the following line of input came originally from source file *filename* and its line number there was *linenum*. Keep in mind that *filename* is not just a file name: it is surrounded by doublequote characters so that it looks like a string constant.

`#line` *anything else*

anything else is checked for macro calls, which are expanded. The result should be a decimal integer constant followed optionally by a string constant, as described above.

'`#line`' commands alter the results of the '`__FILE__`' and '`__LINE__`' predefined macros from that point on. See section 1.4.3.1 [Standard Predefined], page 10.

1.7 Miscellaneous Preprocessor Commands

This section describes two additional preprocesor commands. They are not very useful, but are mentioned for completeness.

The *null command* consists of a '#' followed by a Newline, with only whitespace (including comments) in between. A null command is understood as a preprocessor command but has no effect on the preprocessor output. The primary significance of the existence of the null command is that an input line consisting of just a '#' will produce no output, rather than a line of output containing just a '#'. Supposedly some old C programs contain such lines.

The '`#pragma`' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '`#pragma`' first attempts to run the game `rogue`; if that fails, it tries to run the game `hack`; if that fails, it tries to run GNU Emacs displaying the Tower of Hanoi; if that fails, it reports a fatal error. In any case, preprocessing does not continue.

1.8 C Preprocessor Output

The output from the C preprocessor looks much like the input, except that all preprocessor command lines have been replaced with blank lines and all comments with spaces. Whitespace within a line is not altered; however, a space is inserted after the expansions of most macro calls.

Source file name and line number information is conveyed by lines of the form

```
# linenum filename
```

which are inserted as needed into the middle of the input (but never within a string or character constant). Such a line means that the following line originated in file *filename* at line *linenum*.

1.9 Invoking the C Preprocessor

Most often when you use the C preprocessor you will not have to invoke it explicitly: the C compiler will do so automatically. However, the preprocessor is sometimes useful individually.

The C preprocessor expects two file names as arguments, *infile* and *outfile*. The preprocessor reads *infile* together with any other files it specifies with '`#include`'. All the output generated by

the combined input files is written in *outfile*.

Either *infile* or *outfile* may be '-', which as *infile* means to read from standard input and as *outfile* means to write to standard output. Also, if *outfile* or both file names are omitted, the standard output and standard input are used for the omitted file names.

Here is a table of command options accepted by the C preprocessor. Most of them can also be given when compiling a C program; they are passed along automatically to the preprocessor when it is invoked by the compiler.

'`-P`' Inhibit generation of '`#`'-lines with line-number information in the output from the preprocessor (see section 1.8 [Output], page 32). This might be useful when running the preprocessor on something that is not C code and will be sent to a program which might be confused by the '`#`'-lines

'`-C`' Do not discard comments: pass them through to the output file. Comments appearing in arguments of a macro call will be copied to the output before the expansion of the macro call.

'`-T`' Process ANSI standard trigraph sequences. These are three-character sequences, all starting with '`??`', that are defined by ANSI C to stand for single characters. For example, '`??/`' stands for '\', so '`'??/n'`' is a character constant for Newline. Strictly speaking, the GNU C preprocessor does not support all programs in ANSI Standard C unless '`-T`' is used, but if you ever notice the difference it will be with relief.

You don't want to know any more about trigraphs.

'`-pedantic`'

Issue warnings required by the ANSI C standard in certain cases such as when text other than a comment follows '`#else`' or '`#endif`'.

'`-I` *directory*'

Add the directory *directory* to the end of the list of directories to be searched for header files (see section 1.3.2 [Include Syntax], page 4). This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '`-I`' option, the directories are scanned in left-to-right order; the standard system directories come after.

'`-I-`' Any directories specified with '`-I`' options before the '`-I-`' option are searched only for the case of '`#include "`*file*`"`'; they are not searched for '`#include <`*file*`>`'.

If additional directories are specified with '`-I`' options after the '`-I-`', these directories are searched for all '`#include`' directives.

In addition, the '`-I-`' option inhibits the use of the current directory as the first search directory for '`#include "`*file*`"`'. Therefore, the current directory is searched only if it is requested explicitly with '`-I.`'. Specifying both '`-I-`' and '`-I.`' allows you to control

precisely which directories are searched before the current one and which are searched after.

'-nostdinc'
: Do not search the standard system directories for header files. Only the directories you have specified with '-I' options (and the current directory, if appropriate) are searched.

'-D *name*' Predefine *name* as a macro, with definition '1'.

'-D *name*=*definition*'
: Predefine *name* as a macro, with definition *definition*. There are no restrictions on the contents of *definition*, but if you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

'-U *name*' Do not predefine *name*. If both '-U' and '-D' are specified for one name, the '-U' beats the '-D' and the name is not predefined.

'-undef' Do not predefine any nonstandard macros.

'-d' Instead of outputting the result of preprocessing, output a list of '#define' commands for all the macros defined during the execution of the preprocessor.

'-M' Instead of outputting the result of preprocessing, output a rule suitable for **make** describing the dependencies of the main source file. The preprocessor outputs one **make** rule containing the object file name for that source file, a colon, and the names of all the included files. If there are many included files then the rule is split into several lines using '\'-newline.

This feature is used in automatic updating of makefiles.

'-MM' Like '-M' but mention only the files included with '#include "*file*"'. System header files included with '#include <*file*>' are omitted.

'-i *file*' Process *file* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *file* is discarded, the only effect of '-i *file*' is to make the macros defined in *file* available for use in the main input.

Concept Index

C

cascaded macros 24
commands 3
concatenation 14
conditionals 25

H

header file 3

L

line control 31

M

macro body uses macro 24

N

null command 32

O

options 33
output format 32

P

predefined macros 10
preprocessor commands 3

R

redefining macros 17

S

self-reference 22
semicolons (after macro calls) 19
side effects (in macro arguments) 21
stringification 13

U

undefining macros 16
unsafe macros 21

Index of Commands, Macros and Options

#

#elif . . . 27
#else . . . 27
#error . . . 30
#if . . . 26
#ifdef . . . 29
#ifndef . . . 29
#include . . . 4
#line . . . 31
#pragma . . . 32

-

-C . . . 33
-d . . . 34
-D . . . 34
-i . . . 34
-I . . . 33
-M . . . 34
-MM . . . 34
-P . . . 33
-pedantic . . . 33
-T . . . 33
-U . . . 34
-undef . . . 34

_

__DATE__ . . . 11

_

__FILE__ . . . 11

_

__LINE__ . . . 11

_

__STDC__ . . . 11

_

__TIME__ . . . 11

B

BSD . . . 12

D

defined . . . 29

M

M68020 . . . 12
m68k . . . 12
mc68000 . . . 12

N

ns32000 . . . 13

P

pyr . . . 13

S

sequent . . . 13
sun . . . 13
system header files . . . 4

U

unix . . . 12

V

vax . . . 12